Specification and Generation of Environment for Model Checking of Software Components

Pavel Parizek [a,1], Frantisek Plasil [a,b,1]

[a] *Department of Software Engineering*
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{*parizek, plasil*} @ *nenya.ms.mff.cuni.cz*

[b] *Institute of Computer Science*
Academy of Sciences of the Czech Republic
Prague, Czech Republic
plasil @ cs.cas.cz

Abstract

Model checking of isolated software components is inherently not possible because a component does not form a complete program with an explicit starting point. To overcome this obstacle, it is typically necessary to create an environment of the component which is the intended subject to model checking. We present our approach to automated environment generation that is based on behavior protocols [9]; to our knowledge, this is the only environment generator designed for model checking of software components. We compare it with the approach taken in the Bandera Environment Generator tool [12], designed for model checking of sets of Java classes.

Key words: Software components, behavior protocols, model checking, automated generation of environment

1 Introduction

Model checking is one of the approaches to formal verification of software systems that gets a lot of attention at present. Still, there are some obstacles that have to be addressed, at least partially, before model checking of software can be widely used in practice. Probably the biggest problem is the size of state space typical for software systems. One solution to this problem (state explosion) is the decomposition of a software system into small and well-defined units, components.

[1] This work was partially supported by the Czech Academy of Sciences (project 1ET400300504) and France Telecom under the external research contract number 46127110.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

Nevertheless, a component usually cannot be checked in isolation, because it does not form a complete program inherently needed to apply model checking. It is, therefore, necessary to create a model of the environment of the component subject to model checking, and then check the whole program, composed of the environment and component. The environment should be created in a way that minimizes the increase of the state space size caused by the composition.

1.1 Goals and Structure of the Paper

The paper aims at addressing the problem of automated generation of environment for model checking of software components implemented in the Java language. The main goal is to present our approach that is based on behavior protocols [9] and to compare it with the approach taken in the Bandera Environment Generator tool [12], which is the only other Java focused approach we are aware of.

The remainder of the paper is organized as follows. Sect. 2 provides an example to illustrate the problem of environment generation and Sect. 3 introduces the Bandera Environment Generator (BEG) [12]. Sect. 4 starts with an overview of behavior protocols [9] and then presents the key contribution - the description of our approach to specification and generation of environment based on behavior protocols. Sect. 5 provides comparison of the two approaches and briefly mentions our proof of concept implementation. The rest of the paper contains related work and a conclusion.

2 Motivation

In order to illustrate how an environment can be created, we present a simple example - a Java class `DatabaseImpl` and a handwritten environment for this class, assuming `DatabaseImpl` is the intended subject to model checking. The class implements one interface and requires one internal reference of an interface type to be set. Therefore, it can be also looked upon as a `Database` component with one provided and one required interface.

Key fragments of source code of the `DatabaseImpl` class look as follows:

```
public interface IDatabase {
  public void start();
  public void stop();
  public void insert(int key, String value);
  public String get(int key);
}

public class DatabaseImpl implements IDatabase {
  private ILogger log;
```

```
  public void start() {
    log.start();
  }

  public void stop() {
    log.stop();
  }

  public void insert(int key, String value) {
    ...
  }

  public String get(int key) {
    ...
  }
}
```

In general, an environment should allow the model checker (i) to search for concurrency errors (typically reflected by introducing several threads that are executed in parallel), and (ii) to check all the control flow paths (usually addressed by a random choice of parameter values for all methods).

Captured by "the important" fragments of its source code, such environment could take the following form:

```
public class EnvThread extends Thread {
  IDatabase db;
  ...

  public void run() {
    db.insert(getRandomInt(), getRandomString());
    String val = db.get(getRandomInt());
    ...
  }
}

public static void main(String[] args) {
  IDatabase db = new DatabaseImpl();
  db.setLogger(new LoggerImpl());

  db.start();

  new EnvThread(db).start();
  new EnvThread(db).start();
  ...
```

```
  db.stop();
}
```

In the example, two threads of control, which enable the model checker to search for concurrency errors, are created. A random choice of parameter values for the purpose of checking all the control flow paths is employed as well (`getRandom...` calls).

Obviously, creating an environment by hand is hard and tedious work even in simple cases. A straightforward solution to this problem is to automatically generate the environment from a higher-level abstraction than the code provides. In Sect. 3 and 4, we present two solutions based on this idea.

3 Environment Generator in Bandera

3.1 Bandera

Bandera [6] is a tool set designed for model checking of complete Java programs, i.e. those featuring a main method. It is composed of several modules - model extractor, model translator, environment generator, and model checker, to name the key of them. The model extractor extracts a (finite) internal model from Java source code and the model translator translates the internal model into the input language of a target model checker. Here, the Bandera tool set supported the Spin and Java PathFinder model checkers originally, but currently it is intended mainly for a Bandera specific model checker (Bogor [11]).

3.2 Bandera Environment Generator

The Bandera Environment Generator (BEG) [12] is a tool for automated generation of environment for Java classes. Given a complete Java program, the user of the BEG tool has to decompose the program into two parts - the tested *unit*, i.e. the classes to be tested, and its *environment*. Since the environment part is usually too complex for the purpose of model checking, it is necessary to create an abstract environment. This abstract environment can be generated from a model created

- either from assumptions the user provided, or
- from a result of code analysis of environment classes (if available).

The model can specify, for example, that a certain method should be called five times in a row, or that it should be executed in parallel with another specific method.

Since, usually, there exist no environment classes in case of software components, we will further consider only the first option - i.e. that the abstract environment is generated from user-specified assumptions. For this purpose, the BEG tool provides two formal notations - LTL and regular expressions.

The actual specification ("environment specification" in the rest of this section) takes the form of program action patterns (method calls, assignments, etc), illustrated below.

An environment specification for the `DatabaseImpl` class presented in Sect. 2, written in the input language of the BEG tool, could be as follows:

```
environment
{
  instantiations
  {
    1 LoggerImpl log;
    IDatabase db = new DatabaseImpl();
    db.setLogger(log);
    int x = 5;
  }

  -- high level specification of the environment behavior
  regular assumptions
  {
    T0: (db.get() | db.insert())*
    T1: (db.get(x) | db.insert(5, "abcd"))*
  }
}
```

The `instantiations` section allows the user to specify how many instances of a certain type should be created and under which names they can be referenced. In this example, two objects are instantiated - the `log` instance of the `LoggerImpl` class and the `db` instance of the `DatabaseImpl` class.

The `regular assumptions` section contains regular expressions describing the behavior of the environment with respect to the tested classes. Each regular expression defines a sequence of actions that should be performed by a single thread of control. In our example, two threads of control are defined, both modeling a sequence of calls to the `insert` and `get` methods on the `IDatabase` interface.

Notice that the whole execution is characterized by the specified threads (`T0`, `T1`) - there is no "main" thread. Consequently, calls to the `start` and `stop` methods on the `IDatabase` interface cannot be reasonably modeled in such an environment specification.

The BEG tool also allows to specify parameter values of method calls on the tested classes. If the value of a parameter is not specified, as in the thread `T0` above, then it is non-deterministically selected from all the available values of a given type (e.g. from all allocated instances of a given class in the case of a reference type) during model checking. As a parameter to a method call, it is even possible to use a variable defined in the `instantiations` section (such as `x` above).

As the BEG tool is not intended specifically for software components, but rather for plain Java classes, it is necessary to manually specify the environment for the classes that implement a target component; an alternative would be to develop a tool for automatic translation of an ADL specification of the component's architecture and behavior into the input language of the BEG tool.

However, since the most recent Bandera release is an alpha version only [6], not being fully stable yet, we have decided to use the Java PathFinder model checker (JPF) [13]. Consequently, we faced the problem to create an environment generator, since none was available (BEG is not intended for components and, moreover, the latest Bandera version does not allow to use the Java PathFinder as a target model checker any more).

4 Environment Generator for Java PathFinder

We have built our own environment generator for model checking of components implemented in the Java language. Our approach stems from the assumption that components are during design specified in an ADL (Architecture Description Language), which, in particular, includes specification of their provided and required interfaces and also specification of their behavior. The latter is done via behavior protocols [9]. In this section we show how this behavior specification can be advantageously employed for generating an environment necessary for component model checking.

4.1 Behavior protocols

A behavior protocol is an expression that describes the behavior of a software component in terms of atomic events on the provided and required interfaces of the component, i.e. in terms of accepted and emitted method call requests and responses on those interfaces.

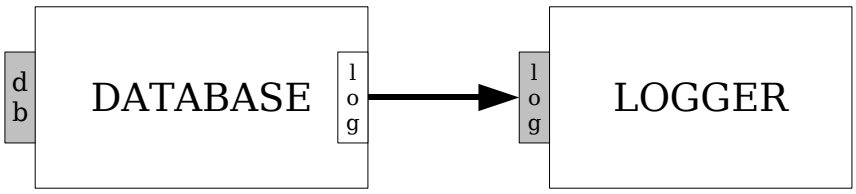

Fig. 1. The DATABASE and LOGGER components, defined in Sect. 2

A protocol example for the Database component from Fig. 1 is below:

```
?db.start↑ ; !log.start ; !db.start↓ ; (?db.insert || ?db.get)* ;
?db.stop{!log.stop}
```

Since this protocol specifies the interplay on the external interfaces of `Database`, it is its *frame protocol* [9]. Informally speaking, it specifies the `Database` functionality that starts with accepting request for `start` call on `db`. As a reaction it calls `start` at `log` and issues response to the `start` call on `db`. This is followed by accepting `insert` on `db` in parallel with `get` on `db` finitely many times. At the end, it accepts a request for a `stop` call on `db` and, as a reaction, it calls `stop` at `log` and issues response to the `stop` call on `db`.

Each event has the following syntax: `<prefix><interface>.<method> <suffix>` (where the suffix is optional; the events having no suffix are syntactical shortcuts explained below). The prefix `?` denotes an *accept* event and the prefix `!` denotes an *emit* event. The suffix ↑ stands for a request (i.e. a method call) and the suffix ↓ stands for a response (i.e. return from a method). An expression of the form `!i.m` is a shortcut for `!i.m↑;?i.m↓`, an expression of the form `?i.m` is a shortcut for `?i.m↑;!i.m↓` and an expression of the form `?i.m{prot}` is a shortcut for `?i.m↑;prot;!i.m↓`, where `prot` is a protocol. The `NULL` keyword denotes an empty protocol.

The example above presents also several operators. The `;` character is the sequence operator, `*` is the repetition operator and `||` is the or-parallel operator. Behavior protocols support also an alternative operator `+` and an and-parallel operator `|`. In fact, the or-parallel operator is only a shortcut; e.g. `a || b` stands for `a + b + (a | b)`. The `|` operator denotes all the possible interleavings of traces that correspond to its operands.

A behavior protocol defines a possibly infinite set of event traces, each of them being finite.

Each component has a frame protocol associated with it, and a composite component can have also an architecture protocol [9]. The frame protocol of a component describes its external behavior, what means that it can contain only the events on external interfaces of the component. On the other hand, the architecture protocol describes the behavior of a component in terms of composition of its subcomponents at the first level of nesting.

4.2 Cooperation of Java PathFinder with the Protocol Checker

When checking a component application specified via ADL with behavior protocols, it is necessary (i) for each composite component in the hierarchy to check compositional compliance of subcomponents at the first level of nesting and also compliance of a frame protocol with an architecture protocol (ii) and for each primitive component to verify that an implementation of the component obeys its frame protocol. For the purpose of checking compliance of protocols, we use the protocol checker [7] developed in our research group, and for checking that a primitive component obeys its frame protocol, we

use a tool created via cooperation of JPF with our protocol checker [8]. The tool has to be applied to a program composed of a target component and its environment.

Communication between JPF and the protocol checker during checking of the `Database` component is depicted on Fig. 2. The left part of the schema shows the JPF traversing the code (state space) of the component and the right part shows the state space of the protocol checker, which is determined by the frame protocol of the component. A plugin for JPF, which we have developed, traces execution of the invoke and return instructions that are related to methods of the provided and required interfaces of a target component, and notifies the protocol checker of those instructions in the form of atomic request and response events. The protocol checker verifies that the trace constructed from the received events is compliant with the frame protocol of the component. When the protocol checker encounters an unexpected event or a missing event, it tells JPF to stop the state space traversal and to report an error (counter example) to the user.

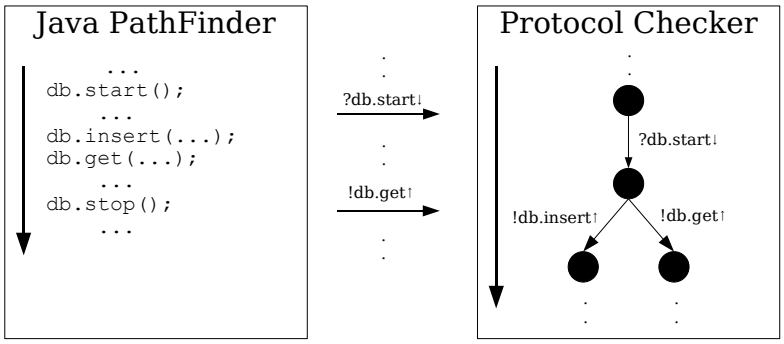

Fig. 2. Communication between the Java PathFinder and Protocol Checker

4.3 Modeling the Environment with Inverted Frame Protocol

The environment of a component can be advantageously modeled by its inverted frame protocol [1], constructed from the components frame protocol by replacing all the accept events with emit events and vice versa. The inverted frame protocol constructed this way forces the environment

- to call a certain method of a particular provided interface of the component at the moment the component expects it, and
- to accept a certain method call issued on a particular required interface of the component at the moment the component "wishes" to do so.

The inverted frame protocol of the `Database` component introduced above is:

```
!db.start↑ ; ?log.start ; ?db.start↓ ; (!db.insert || !db.get)* ;
!db.stop{?log.stop}
```

Our environment generator accepts all syntactically valid frame protocols with the exception of protocols of the form `?a + !b` and `!a* ; ?b`. The reason for not supporting frame protocols of the form `?a + !b` is that the environment driven by inversion of such a protocol cannot determine how long it should wait for the `!b` event to occur before it emits a call that corresponds to the `?a` event and therefore disables the other alternative (i.e. `!b`). Protocols of the form `!a* ; ?b` are not supported for a similar reason - the environment is not able to determine when the repetition `!a*` is going to finish. It is recommended to use protocols of the form `!a* ; !b` instead (wherever possible) because in such case the `!b` event tells the environment that the repetition has finished.

In order to minimize the size of the state space that JPF has to traverse, our environment generator performs several transformations of the frame protocol of the target component before creating the inverted frame protocol and generating the code of the environment. The key goal of the transformations is to

- get as many instances of the alternative operator `+` as possible at the outermost level of protocol nesting. The advantage of this approach is that all these alternatives can be checked in parallel by multiple instances of JPF, thus lowering the time requirements for model checking of the target component.
- reduce the number of repetitions, and also event interleavings caused by the `|` operator, even at the cost of accuracy.

For example, our generator transforms

- an iteration over some subprotocol to an alternative between an empty protocol and a sequence of two copies of the subprotocol (e.g. the protocol `!a*` is transformed to the protocol `NULL + (!a ; !a)`),
- a sequence that contains some alternatives to an alternative between all possible sequences (e.g. the protocol `!a ; (!b1 + !b2)` is transformed to the protocol `(!a ; !b1) + (!a ; !b2)`),
- an and-parallel operator connecting two subprotocols, both of them being alternatives, to an alternative between selected pairs of subprotocols connected by the `|` operator - the pairs are selected in a way ensuring that each element of the two alternatives is present at least in one of the pairs (e.g. the protocol `(!a1 + !a2) | (!b1 + !b2)` is transformed to the protocol `(!a1 | !b1) + (!a2| !b2)`), and
- an and-parallel operator with three or more subprotocols to an alternative between selected pairs of subprotocols, where each pair is connected by the `|` operator and followed by a sequence of subprotocols that do not belong into the selected pair; the pairs are selected in such a way that the first subprotocol is paired with the second, the second with the third, and so on

(e.g the protocol `a | b | c | d` is transformed to the protocol `((a | b) ; c ; d) + ((b | c) ; a ; d) + ((c | d) ; a ; b)`).

4.4 Specification of Values of Method Parameters

Our solution to specification of the possible values of method parameters is based on the idea that the user defines the set of values which are to be considered as parameters. From the implementation point of view, these sets are to be put into a special Java class serving as a container for all the sets of values. The value of a method parameter of certain type is later non-deterministically selected from the set of values considered for that type and method. In addition to the sets of values common for the whole component, it is also possible to define sets that are specific to a particular method or interface.

Below is a fragment of the specification of values for the `Database` component:

```
putIntSet("IDatabase", "insert", new int[]{1, 2, 5, 10});
putIntSet("", "", new int[]{1, 3, 5, 12});
putStringSet("", "", new String[]{"abcd", "EFGH1234"});
```

The first statement defines a set of integer values that is specific to the `insert` method of the `IDatabase` interface. The other two statements define the sets of integers and strings that are to be applied to all methods of the `Database` component interfaces.

The main drawback of this approach is that the user has to define on his/her own the sets of values in such a way that will force the model checker to check all the control flow paths in the component.

5 Evaluation

In this section we compare the two approaches to modeling the environment described above, i.e. the approach of the BEG tool and our approach based on behavior protocols.

The main differences between them are:

- The BEG tool allows to specify parallelism only at the outermost level of regular expressions that specify behavior of the environment (there is no such limit in case of behavior protocols).
- Behavior protocols have no support for method parameters, therefore the possible values of method parameters must be specified separately in a special Java class, while the BEG tool allows to specify the values of method parameters directly in the expressions that specify behavior of the environment.

It is worth to mention that there is also a difference in that the BEG tool targets plain Java classes with informally specified provided and required in-

terfaces, while our approach targets the software components having provided and required interfaces defined in an explicit way.

As a speciality, another advantage of support for specification of parameter values directly in expressions that specify behavior is that it enables the environment generator to select a proper version of an overloaded method - or to generate a code that will non-deterministically invoke all versions of the method that conform to the specification.

We have created an implementation of the environment generator that uses the inverted frame protocol of a component as a model of the environments behavior. It aims at components that use the Fractal Component Model [5] and expects that the Fractal ADL is used to define components. We have successfully applied our environment generator to a component-based application composed of 20 components. Transformations of the frame protocol, described in Sect. 4.3, reduce the size of the state space determined by the protocol approximately thirty times in case of more complex components, therefore lowering also the time required for model checking of the components, all that at the cost of accuracy, though. Nevertheless, model checking of more complex components with environment generated from their frame protocols with no transformations applied is not feasible. Despite the abstractions of the environment introduced by transformations of the frame protocol, the technique is still much more systematic than simple testing. Let us again emphasize that model checking of a component without an environment is not possible at all, because JPF is applicable only to complete Java programs, not isolated software components.

6 Related work

Except for the Bandera Environment Generator [12], we are not aware of any other approach to specification and generation of environment for model checking of software components or parts of object-oriented programs. Nevertheless, there exist model checkers for object-oriented programs that do not need to generate an environment because these tools usually extract a finite model from a complete program (featuring the main method) and then check the model - an example of such a model checker is Zing [2].

There are also tools that solve the problem of automatic generation of environment for fragments of procedural programs (e.g. drivers, libraries, etc). An example of such a tool is the SLAM [4] model checker, which is a part of the SDV tool for verification of device drivers for the Windows operating system. Given a program, the checker creates a Boolean abstraction of the program (all value types approximated by Boolean) and then checks whether some desired temporal properties hold for the abstraction. It uses the principle of refinement to discard errors that are present in the abstraction but not in the original program (false negatives). The environment for device drivers is defined by the interfaces provided by the Windows kernel. The SLAM tool

models the environment via training [3]. Here, the basic principle is that, for a certain procedure P that is to be modeled, it first takes several drivers that use the procedure P, then it runs the SDV tool on those drivers and therefore gets several Boolean abstractions of the procedure P, and finally merges all those abstractions and puts the resulting Boolean abstraction of the kernel procedure P into a library for future reuse.

Our tool for environment generation is partially based on [10]. The tool that is described in the thesis, designed for the Bandera tool set, also uses the inverted frame protocol idea; it is also focused on components compliant to the Fractal Component Model [5]. We decided not to use this tool mainly because it generates an environment that increases the state space size quite significantly, since it does not employ any of transformations described in Sect. 4.3 and also does not provide any means for specification of method parameter values - all that makes it almost unusable in practice.

7 Conclusion

Direct model checking of isolated software components is usually not possible because model checkers can handle only complete programs. Therefore, it is necessary to create an environment for each component subject to model checking.

In this paper, we have compared two approaches to generating environment of components, resp. classes - namely the Bandera Environment Generator (BEG) tool [12] in Sect. 3, and our approach that is based on behavior protocols [9] in Sect. 4. Main differences between the two approaches lie in the level of support for parallelism, in support for specification of parameter values, and in the fact that the BEG tool is focused on plain Java classes while our approach targets software components with explicitly defined provided and required interfaces.

As to future work, an automated derivation of sets of values used for non-deterministic choice of method parameters is our current goal. It is motivated by the fact that manual definition of such sets requires the user to carefully capture a way that will let the model checker to check all the control flow paths in a target component. A viable approach to the derivation of possible parameter values could be to use static analysis of Java source code (or byte code).

Acknowledgments

We would like to record a special credit to Jiri Adamek and Nicolas Rivierre for valuable comments and Jan Kofron also for many hints regarding the integration of the protocol checker with JPF.

References

[1] Adamek, J., and F. Plasil, *Component Composition Errors and Update Atomicity: Static Analysis*, Journal of Software Maintenance and Evolution: Research and Practice, **17**(2005), pp. 363-377

[2] Andrews, T., S. Qadeer, S. K. Rajamani, J. Rehof and Y. Xie, *Zing: a model checker for concurrent software*, Technical report, Microsoft Research, 2004

[3] Ball, T., V. Levin and F. Xie, *Automatic Creation of Environment Models via Training*, TACAS 2004, 93-107

[4] Ball, T., S. K. Rajamani, *The SLAM Project: Debugging System Software via Static Analysis*, POPL 2002, ACM, 1-3

[5] Bruneton, E., T. Coupaye, M. Leclercq, V. Quma, and J. B. Stefani, *An Open Component Model and its Support in Java*, In Proceedings of the International Symposium on Component-based Software Engineering (ICSE 2004 - CBSE7), LNCS, **3054**(2004), May 2004

[6] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zhueng, *Bandera: Extracting Finite-state Models from Java Source Code*, ICSE 2000, ACM, 439-448

[7] Mach, M., F. Plasil and J. Kofron, *Behavior Protocol Verification: Fighting State Explosion*, International Journal of Computer and Information Science, **6**(2005), 22-30

[8] Parizek, P., F. Plasil and J. Kofron, *Model Checking of Software Components: Making Java PathFinder Cooperate with Behavior Protocol Checker*, Tech. Report No. 2006/2, Dep. of SW Engineering, Charles University, Jan 2006

[9] Plasil, F., and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, **28**(2002)

[10] Potrusil, T., "Specifying Missing Component Environment in Bandera", Master Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2005

[11] Robby, M. Dwyer and J. Hatcliff, *Bogor: An extensible and highly-modular model checking framework*, In FSE 03: Foundations of Software Engineering, pp. 267-276, ACM, 2003

[12] Tkachuk, O., M. B. Dwyer and C. S. Pasareanu, *Automated Environment Generation for Software Model Checking*, 18th IEEE International Conference on Automated Software Engineering (ASE03), p. 116, 2003

[13] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model Checking Programs*, Automated Software Engineering Journal, **10**(2003)